下载APP

21 | 函数：请叫我“一等公民”

2021-11-29 Tony Bai

《Tony Bai · Go语言第一课》 课程介绍 >

讲述：Tony Bai
时长 26:33 大小 24.32M

你好，我是 Tony Bai。

在前面的几讲中，我们学习了用于对现实世界实体抽象的类型，以及用来实现算法逻辑控制的几种控制结构。从这一讲开始，我们来学习一下 Go 代码中的基本功能逻辑单元：**函数**。

学到这里，相信你对 Go 中的函数已经不陌生了，因为我们在前面的示例程序中一直都在使用函数。函数是现代编程语言的基本语法元素，无论是在命令式语言、面向对象语言还是动态脚本语言中，函数都位列 C 位。

Go 语言也不例外。在 Go 语言中，**函数是唯一一种基于特定输入，实现特定任务并可返回任务执行结果的代码块**（Go 语言中的方法本质上也是函数）。如果忽略 Go 包在 Go 代码

组织层面的作用，我们可以说 **Go 程序就是一组函数的集合**，实际上，我们日常的 Go 代码编写大多都集中在实现某个函数上。

但“一龙生九子，九子各不同”！虽然各种编程语言都加入了函数这个语法元素，但各个语言中函数的形式与特点又有不同。那么 Go 语言中函数又有哪些独特之处呢？考虑到函数的重要性，我们会用三节课的时间，全面系统地讲解 Go 语言的函数。

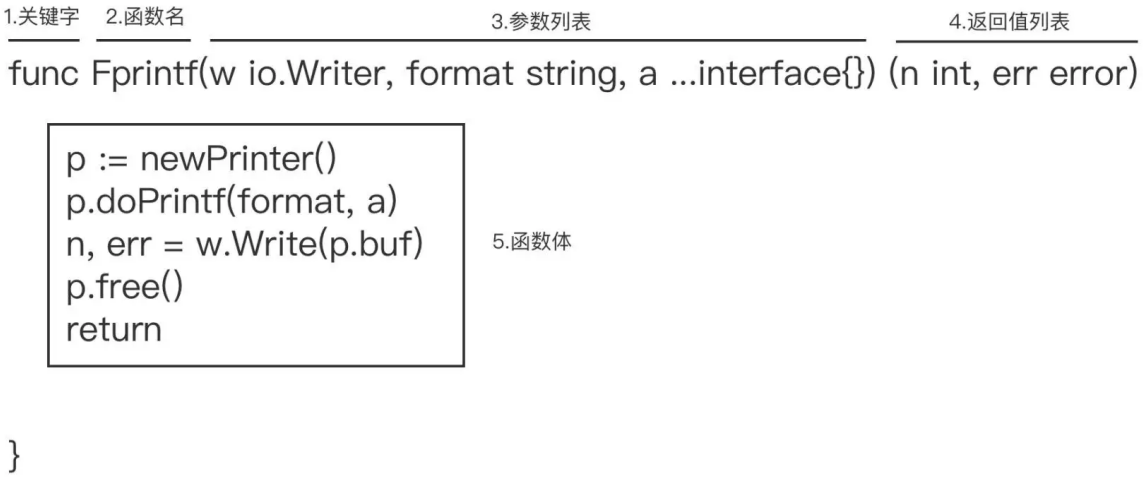
在这一节课中，我们就先来学习一下函数基础，以及 Go 函数最与众不同的一大特点。我们先从最基本的函数声明开始说起。

Go 函数与函数声明

函数对应的英文单词是 **Function**，Function 这个单词原本是**功能、职责**的意思。编程语言使用 Function 这个单词，表示将一个大问题分解后而形成的、若干具有特定功能或职责的小任务，可以说十分贴切。函数代表的小任务可以在一个程序中被多次使用，甚至可以在不同程序中被使用，因此**函数的出现也提升了整个程序界代码复用的水平**。

那 Go 语言中，函数相关的语法形式是怎样的呢？我们先来看最常用的 Go 函数声明。

在 Go 中，我们定义一个函数的最常用方式就是使用**函数声明**。我们以 Go 标准库 fmt 包提供的 Fprintf 函数为例，看一下一个**普通 Go 函数的声明**长啥样：



我们看到一个 Go 函数的声明由五部分组成，我们一个个来拆解一下。

第一部分是关键字 func，Go 函数声明必须以关键字 func 开始。

第二部分是函数名。函数名是指代函数定义的标识符，函数声明后，我们会通过函数名这个标识符来使用这个函数。在同一个 Go 包中，函数名应该是唯一的，并且它也遵守 Go 标识符的导出规则，也就是我们之前说的，首字母大写的函数名指代的函数是可以在包外使用的，小写的就只在包内可见。

第三部分是参数列表。参数列表中声明了我们将要在函数体中使用的各个参数。参数列表紧接在函数名的后面，并用一个括号包裹。它使用逗号作为参数间的分隔符，而且每个参数的参数名在前，参数类型在后，这和变量声明中变量名与类型的排列方式是一致的。

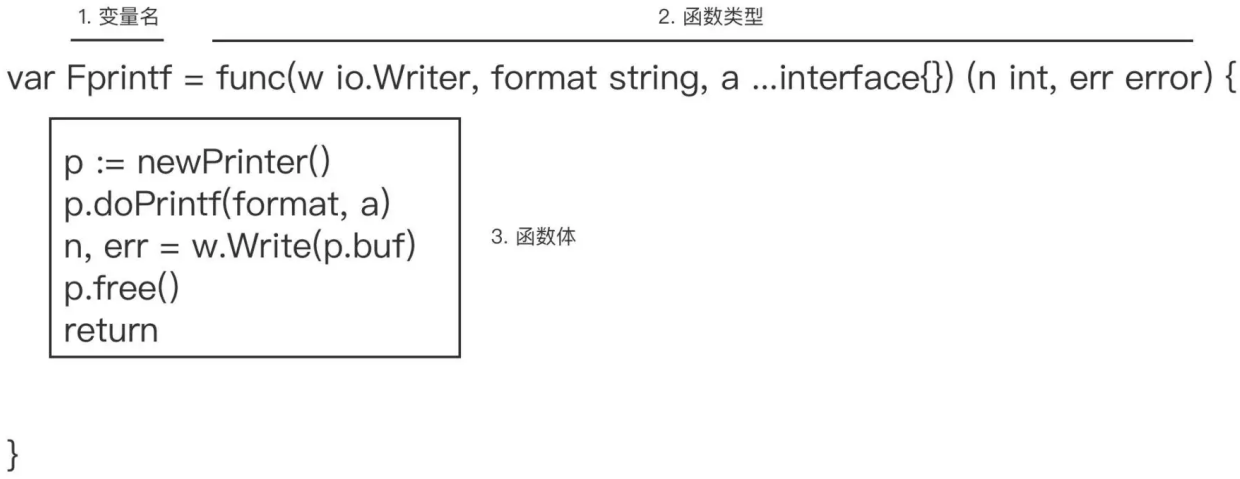
另外，Go 函数支持变长参数，也就是一个形式参数可以对应数量不定的实际参数。Fprintf 就是一个支持变长参数的函数，你可以看到它第三个形式参数 a 就是一个变长参数，而且变长参数与普通参数在声明时的不同点，就在于它会在类型前面增加了一个“...”符号。关于函数对变长参数的支持，我们在后面还会再讲。

第四部分是返回值列表。返回值承载了函数执行后要返回给调用者的结果，返回值列表声明了这些返回值的类型，返回值列表的位置紧接在参数列表后面，两者之间用一个空格隔开。不过，上图中比较特殊，Fprintf 函数的返回值列表不仅声明了返回值的类型，还声明了返回值的名称，这种返回值被称为**具名返回值**。多数情况下，我们不需要这么做，只需声明返回值的类型即可。

最后，放在一对大括号内的是函数体，函数的具体实现都放在这里。不过，函数声明中的**函数体是可选的**。如果没有函数体，说明这个函数可能是在 Go 语言之外实现的，比如使用汇编语言实现，然后通过链接器将实现与声明中的函数名链接到一起。没有函数体的函数声明是更高级的话题了，你感兴趣可以自己去了解一下，我们这里还是先打好基础。

看到这里，你可能会问：**同为声明，为啥函数声明与之前学过的变量声明在形式上差距这么大呢**? 变量声明中的变量名、类型名和初值与上面的函数声明是怎么对应的呢？

为了让更好地理解函数声明，也给我们后续的讲解做铺垫，这里我们就横向对比一下，把上面的函数声明等价转换为变量声明的形式看看：



转换后的代码不仅和之前的函数声明是等价的，而且这也是完全合乎 Go 语法规则的代码。对照一下这两张图，你是不是有一种豁然开朗的感觉呢？**这不就是在声明一个类型为函数类型的变量吗**！

我们看到，函数声明中的函数名其实就是变量名，函数声明中的 func 关键字、参数列表和返回值列表共同构成了**函数类型**。而参数列表与返回值列表的组合也被称为**函数签名**，它是决定两个函数类型是否相同的决定因素。因此，函数类型也可以看成是由 func 关键字与函数签名组合而成的。

通常，在表述函数类型时，我们会省略函数签名参数列表中的参数名，以及返回值列表中的返回值变量名。比如上面 Fprintf 函数的函数类型是：

复制代码

```
1 func(io.Writer, string, ...interface{}) (int, error)
```

这样，如果两个函数类型的函数签名是相同的，即便参数列表中的参数名，以及返回值列表中的返回值变量名都是不同的，那么这两个函数类型也是相同类型，比如下面两个函数类型：

复制代码

```
1 func (a int, b string) (results []string, err error)
2 func (c int, d string) (sl []string, err error)
```

如果我们把这两个函数类型的参数名与返回值变量名省略，那它们都是func (int, string) ([]string, error)，因此它们是相同的函数类型。

到这里，我们可以得到这样一个结论：**每个函数声明所定义的函数，仅仅是对应的函数类型的一个实例**，就像var a int = 13这个变量声明语句中 a 是 int 类型的一个实例一样。

如果你还记得前面第 17 讲中、使用复合类型字面值对结构体类型变量进行显式初始化的内容，你一定会觉得上面这种、用变量声明来声明函数变量的形式，似曾相识，我们把这两种形式都以最简化的样子表现出来，看下面代码：

复制代码

```
1 s := T{}       // 使用复合类型字面值对结构体类型T的变量进行显式初始化
2 f := func(){} // 使用变量声明形式的函数声明
```

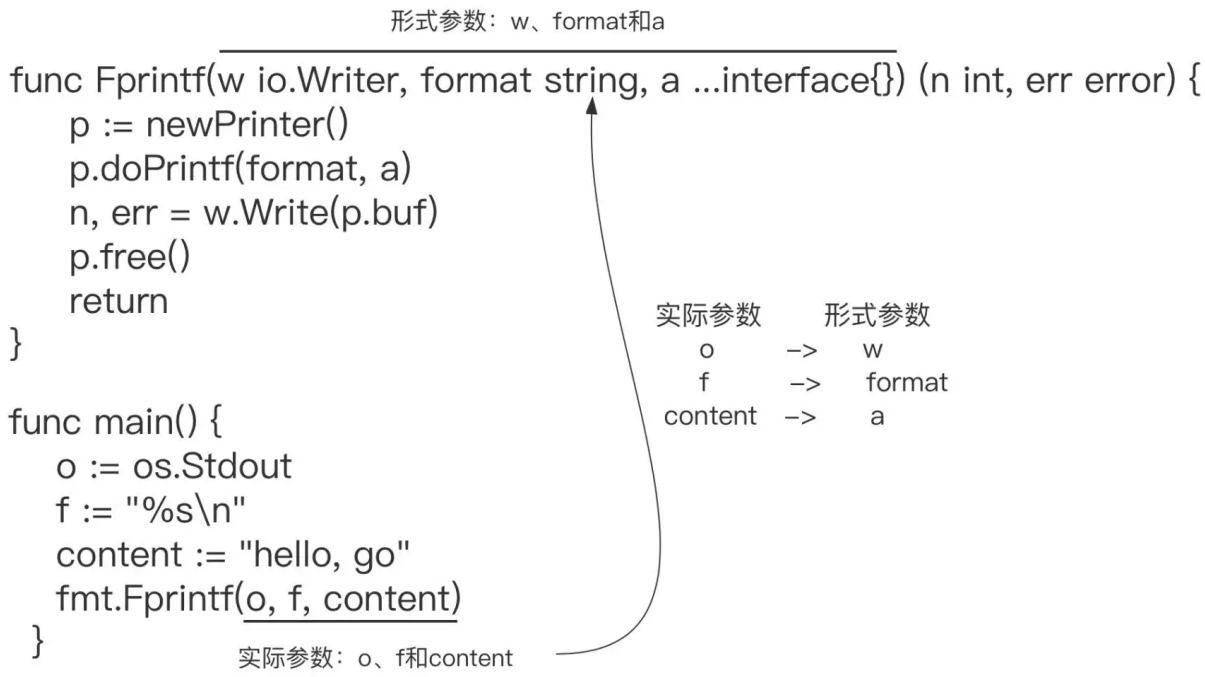
这里，T{}被称为复合类型字面值，那么处于同样位置的 func(){}是什么呢？Go 语言也为它准备了一个名字，叫“**函数字面值**（Function Literal）”。我们可以看到，函数字面值由函数类型与函数体组成，它特别像一个没有函数名的函数声明，因此我们也叫它**匿名函数**。匿名函数在 Go 中用途很广，稍后我们会细讲。

讲到这里，你可能会想：既然是等价的，那我以后就用这种变量声明的形式来声明一个函数吧。万万不可！我这里只是为了帮你理解函数声明做了一个等价变换。在 Go 中的绝大多数情况，我们还是会通过传统的函数声明来声明一个特定函数类型的实例，也就是我们俗称的“定义一个函数”。

好了，横向对比就到此为止了，现在我们继续回到函数声明中来， 详细看看函数声明的重要组成部分——参数。

函数参数的那些事儿

函数参数列表中的参数，是函数声明的、用于函数体实现的局部变量。由于函数分为声明与使用两个阶段，在不同阶段，参数的称谓也有不同。在函数声明阶段，我们把参数列表中的参数叫做**形式参数**（Parameter，简称形参），在函数体中，我们使用的都是形参；而在函数实际调用时传入的参数被称为**实际参数**（Argument，简称实参）。为了便于直观理解，我绘制了这张示意图，你可以参考一下：



当我们实际调用函数的时候，实参会传递给函数，并和形式参数逐一绑定，编译器会根据各个形参的类型与数量，来检查传入的实参的类型与数量是否匹配。只有匹配，程序才能继续执行函数调用，否则编译器就会报错。

Go 语言中，函数参数传递采用是**值传递**的方式。所谓“值传递”，就是将实际参数在内存中的表示**逐位拷贝**（Bitwise Copy）到形式参数中。对于像整型、数组、结构体这类类型，它们的内存表示就是它们自身的数据内容，因此当这些类型作为实参类型时，值传递拷贝的就是它们自身，传递的开销也与它们自身的大小成正比。

但是像 string、切片、map 这些类型就不是了，它们的内存表示对应的是它们数据内容的“描述符”。当这些类型作为实参类型时，值传递拷贝的也是它们数据内容的“描述符”，不包括数据内容本身，所以这些类型传递的开销是固定的，与数据内容大小无关。这种只拷贝“描述符”，不拷贝实际数据内容的拷贝过程，也被称为**“浅拷贝”**。

不过函数参数的传递也有两个例外，当函数的形参为接口类型，或者形参是变长参数时，简单的值传递就不能满足要求了，这时 Go 编译器会介入：对于类型为接口类型的形参，Go 编译器会把传递的实参赋值给对应的接口类型形参；对于为变长参数的形参，Go 编译器会将零个或多个实参按一定形式转换为对应的变长形参。

那么这里，零个或多个传递给变长形式参数的实参，被 Go 编译器转换为何种形式了呢？我们通过下面示例代码来看一下：

复制代码

```
func myAppend(sl []int, elems ...int) []int {
    fmt.Printf("%T\n", elems) // []int
    if len(elems) == 0 {
        println("no elems to append")
        return sl
    }

    sl = append(sl, elems...)
    return sl
}

func main() {
    sl := []int{1, 2, 3}
    sl = myAppend(sl) // no elems to append
    fmt.Println(sl) // [1 2 3]
    sl = myAppend(sl, 4, 5, 6)
    fmt.Println(sl) // [1 2 3 4 5 6]
}
```

我们重点看一下代码中的 myAppend 函数，这个函数基于 append，实现了向一个整型切片追加数据的功能。它支持变长参数，它的第二个形参 elems 就是一个变长参数。myAppend 函数通过 Printf 输出了变长参数的类型。执行这段代码，我们将看到变长参数 elems 的类型为[]int。

这也就说明，在 Go 中，**变长参数实际上是通过切片来实现的**。所以，我们在函数体中，就可以使用切片支持的所有操作来操作变长参数，这会大大简化了变长参数的使用复杂度。比如 myAppend 中，我们使用 len 函数就可以获取到传给变长参数的实参个数。

到这里，我们已经学习了函数声明的两个部分。接下来，我们再看看函数声明的最后一部分，返回值列表。

函数支持多返回值

和其他主流静态类型语言，比如 C、C++ 和 Java 不同，Go 函数**支持多返回值**。多返回值可以让函数将更多结果信息返回给它的调用者，Go 语言的错误处理机制很大程度就是建立在多返回值的机制之上的，这个我们在后续课程中还会详细讲解。

函数返回值列表从形式上看主要有三种：

复制代码

```
func foo()                          // 无返回值
func foo() error                    // 仅有一个返回值
func foo() (int, string, error)     // 有2或2个以上返回值
```

如果一个函数没有显式返回值，那么我们可以像第一种情况那样，在函数声明中省略返回值列表。而且，如果一个函数仅有一个返回值，那么通常我们在函数声明中，就不需要将返回值用括号括起来，如果是 2 个或 2 个以上的返回值，那我们还是需要用括号括起来的。

在函数声明的返回值列表中，我们通常会像上面例子那样，仅列举返回值的类型，但我们也可以像 fmt.Fprintf 函数的返回值列表那样，为每个返回值声明变量名，这种带有名字的返回值被称为**具名返回值**（Named Return Value）。这种具名返回值变量可以像函数体中声明的局部变量一样在函数体内使用。

那么在日常编码中，我们究竟该使用普通返回值形式，还是具名返回值形式呢？

Go 标准库以及大多数项目代码中的函数，都选择了使用普通的非具名返回值形式。但在一些特定场景下，具名返回值也会得到应用。比如，当函数使用 defer，而且还在 defer 函数中修改外部函数返回值时，具名返回值可以让代码显得更优雅清晰。关于 defer 的使用，我们会在后面课程中还会细讲。

再比如，当函数的返回值个数较多时，每次显式使用 return 语句时都会接一长串返回值，这时，我们用具名返回值可以让函数实现的可读性更好一些，比如下面 Go 标准库 time 包中的 parseNanoseconds 函数就是这样：

复制代码

```
// $GOROOT/src/time/format.go
func parseNanoseconds(value string, nbytes int) (ns int, rangeErrString string
    if !commaOrPeriod(value[0]) {
        err = errBad
        return
    }
    if ns, err = atoi(value[1:nbytes]); err != nil {
        return
```

```
    }
    if ns < 0 || 1e9 <= ns {
        rangeErrString = "fractional second"
        return
    }

    scaleDigits := 10 - nbytes
    for i := 0; i < scaleDigits; i++ {
        ns *= 10
    }
    return
}
```

了解了上面这些有关 Go 函数的基础知识后，接下来，我们来学习 Go 函数与众不同的一个特点，这个特点使得 Go 函数具有更大的灵活性和表达力。

函数是“一等公民”

这个特点就是，**函数在 Go 语言中属于“一等公民（First-Class Citizen）”**。要知道，并不是在所有编程语言中函数都是“一等公民”。

那么，什么是编程语言的“一等公民”呢？关于这个名词，业界和教科书都没有给出精准的定义。我们这里可以引用一下 wiki 发明人、C2 站点作者沃德·坎宁安 (Ward Cunningham)对“一等公民”的解释：

> 如果一门编程语言对某种语言元素的创建和使用没有限制，我们可以像对待值（value）一样对待这种语法元素，那么我们就称这种语法元素是这门编程语言的“一等公民”。拥有“一等公民”待遇的语法元素可以存储在变量中，可以作为参数传递给函数，可以在函数内部创建并可以作为返回值从函数返回。

基于这个解释，我们来看看 Go 语言的函数作为“一等公民”，表现出的各种行为特征。

特征一：Go 函数可以存储在变量中。

按照沃德·坎宁安对一等公民的解释，身为一等公民的语法元素是可以存储在变量中的。其实，这点我们在前面理解函数声明时已经验证过了，这里我们再用例子简单说明一下：

复制代码

```
var (
```

```
    myFprintf = func(w io.Writer, format string, a ...interface{}) (int, error
        return fmt.Fprintf(w, format, a...)
    }
)

func main() {
    fmt.Printf("%T\n", myFprintf) // func(io.Writer, string, ...interface {})
    myFprintf(os.Stdout, "%s\n", "Hello, Go") // 输出Hello , Go
}
```

在这个例子中，我们把新创建的一个匿名函数赋值给了一个名为 myFprintf 的变量，通过这个变量，我们便可以调用刚刚定义的匿名函数。然后我们再通过 Printf 输出 myFprintf 变量的类型，也会发现结果与我们预期的函数类型是相符的。

特征二：支持在函数内创建并通过返回值返回。

Go 函数不仅可以在函数外创建，还可以在函数内创建。而且由于函数可以存储在变量中，所以函数也可以在创建后，作为函数返回值返回。我们来看下面这个例子：

复制代码

```
func setup(task string) func() {
    println("do some setup stuff for", task)
    return func() {
        println("do some teardown stuff for", task)
    }
}

func main() {
    teardown := setup("demo")
    defer teardown()
    println("do some bussiness stuff")
}
```

这个例子，模拟了执行一些重要逻辑之前的上下文建立（setup），以及之后的上下文拆除（teardown）。在一些单元测试的代码中，我们也经常会在执行某些用例之前，建立此次执行的上下文（setup），并在这些用例执行后拆除上下文（teardown），避免这次执行对后续用例执行的干扰。

在这个例子中，我们在 setup 函数中创建了这次执行的上下文拆除函数，并通过返回值的形式，将这个拆除函数返回给了 setup 函数的调用者。setup 函数的调用者，在执行完对

应这次执行上下文的重要逻辑后，再调用 setup 函数返回的拆除函数，就可以完成对上下文的拆除了。

从这段代码中我们也可以看到，setup 函数中创建的拆除函数也是一个匿名函数，但和前面我们看到的匿名函数有一个不同，这个不同就在于这个匿名函数使用了定义它的函数 setup 的局部变量 task，这样的匿名函数在 Go 中也被称为**闭包**（Closure）。

闭包本质上就是一个匿名函数或叫函数字面值，它们可以引用它的包裹函数，也就是创建它们的函数中定义的变量。然后，这些变量在包裹函数和匿名函数之间共享，只要闭包可以被访问，这些共享的变量就会继续存在。显然，Go 语言的闭包特性也是建立在“函数是一等公民”特性的基础上的，后面我们还会讲解涉及到闭包的内容。

特征三：作为参数传入函数。

既然函数可以存储在变量中，也可以作为返回值返回，那我们可以理所当然地想到，把函数作为参数传入函数也是可行的。比如我们在日常编码时经常使用、标准库 time 包的 AfterFunc 函数，就是一个接受函数类型参数的典型例子。你可以看看下面这行代码，这里通过 AfterFunc 函数设置了一个 2 秒的定时器，并传入了时间到了后要执行的函数。这里传入的就是一个匿名函数：

复制代码

```
1 time.AfterFunc(time.Second*2, func() { println("timer fired") })
```

特征四：拥有自己的类型。

通过我们前面的讲解，你可以知道，作为一等公民的整型值拥有自己的类型 int，而这个整型值只是类型 int 的一个实例，其他作为一等公民的字符串值、布尔值等类型也都拥有自己类型。那函数呢？

在前面讲解函数声明时，我们曾得到过这样一个结论：每个函数声明定义的函数仅仅是对应的函数类型的一个实例，就像`var a int = 13`这个变量声明语句中的 a，只是 int 类型的一个实例一样。换句话说，每个函数都和整型值、字符串值等一等公民一样，拥有自己的类型，也就是我们讲过的**函数类型**。

我们甚至可以基于函数类型来自定义类型，就像基于整型、字符串类型等类型来自定义类型一样。下面代码中的 HandlerFunc、visitFunc 就是 Go 标准库中，基于函数类型进行自定义的类型：

复制代码

```
// $GOROOT/src/net/http/server.go
type HandlerFunc func(ResponseWriter, *Request)

// $GOROOT/sort/genzfunc.go
type visitFunc func(ast.Node) ast.Visitor
```

到这里，我们已经可以看到，Go 函数确实表现出了沃德·坎宁安诠释中“一等公民”的所有特征：Go 函数可以存储在变量中，可以在函数内创建并通过返回值返回，可以作为参数传递给其他函数，可以拥有自己的类型。通过这些分析，你也能感受到，和 C/C++ 等语言中的函数相比，作为“一等公民”的 Go 函数拥有难得的灵活性。

那么在实际生产中，我们怎么才能发挥出这种灵活性的最大效用，帮助我们写出更加优雅简洁的 Go 代码呢？接下来，我们就看几个这方面的例子。

函数“一等公民”特性的高效运用

应用一：函数类型的妙用

Go 函数是“一等公民”，也就是说，它拥有自己的类型。而且，整型、字符串型等所有类型都可以进行的操作，比如显式转型，也同样可以用在函数类型上面，也就是说，**函数也可以被显式转型**。并且，这样的转型在特定的领域具有奇妙的作用，一个最为典型的示例就是标准库 http 包中的 HandlerFunc 这个类型。我们来看一个使用了这个类型的例子：

复制代码

```
func greeting(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "Welcome, Gopher!\n")
}

func main() {
    http.ListenAndServe(":8080", http.HandlerFunc(greeting))
}
```

这我们日常最常见的、用 Go 构建 Web Server 的例子。它的工作机制也很简单，就是当用户通过浏览器，或者类似 curl 这样的命令行工具，访问 Web server 的 8080 端口时，会收到“Welcome, Gopher!”这样的文字应答。我们在 09 讲曾讲过使用 http 包编写 web server 的方法，但当时我没有进一步讲解其中的原理，这一节课中我们就补上这一点。

我们先来看一下 http 包的函数 ListenAndServe 的源码：

复制代码

```
// $GOROOT/src/net/http/server.go
func ListenAndServe(addr string, handler Handler) error {
    server := &Server{Addr: addr, Handler: handler}
    return server.ListenAndServe()
}
```

函数 ListenAndServe 会把来自客户端的 http 请求，交给它的第二个参数 handler 处理，而这里 handler 参数的类型 http.Handler，是一个自定义的接口类型，它的源码是这样的：

复制代码

```
// $GOROOT/src/net/http/server.go
type Handler interface {
    ServeHTTP(ResponseWriter, *Request)
}
```

我们还没有系统学习接口类型，你现在只要知道接口是一组方法的集合就好了。这个接口只有一个方法 ServeHTTP，他的函数类型是`func(http.ResponseWriter, *http.Request)`。这和我们自己定义的 http 请求处理函数 greeting 的类型是一致的，但是我们没法直接将 greeting 作为参数值传入，否则编译器会报错：

复制代码

```
func(http.ResponseWriter, *http.Request) does not implement http.Handler (miss
```

这里，编译器提示我们，函数 greeting 还没有实现接口 Handler 的方法，无法将它赋值给 Handler 类型的参数。现在我们再回过头来看下代码，代码中我们也没有直接将

greeting 传给 ListenAndServe 函数，而是将`http.HandlerFunc(greeting)`作为参数传给了 ListenAndServe。那这个 http.HandlerFunc 究竟是什么呢？我们直接来看一下它的源码：

复制代码

```
// $GOROOT/src/net/http/server.go

type HandlerFunc func(ResponseWriter, *Request)

// ServeHTTP calls f(w, r).
func (f HandlerFunc) ServeHTTP(w ResponseWriter, r *Request) {
        f(w, r)
}
```

通过它的源码我们看到，HandlerFunc 是一个基于函数类型定义的新类型，它的底层类型为函数类型`func(ResponseWriter, *Request)`。这个类型有一个方法 ServeHTTP，然后实现了 Handler 接口。也就是说`http.HandlerFunc(greeting)`这句代码的真正含义，是将函数 greeting 显式转换为 HandlerFunc 类型，后者实现了 Handler 接口，满足 ListenAndServe 函数第二个参数的要求。

另外，之所以`http.HandlerFunc(greeting)`这段代码可以通过编译器检查，正是因为 HandlerFunc 的底层类型是`func(ResponseWriter, *Request)`，与 greeting 函数的类型是一致的，这和下面整型变量的显式转型原理也是一样的：

复制代码

```
type MyInt int
var x int = 5
y := MyInt(x) // MyInt的底层类型为int，类比HandlerFunc的底层类型为func(ResponseWrit
```

应用二：利用闭包简化函数调用。

我们前面讲过，Go 闭包是在函数内部创建的匿名函数，这个匿名函数可以访问创建它的函数的参数与局部变量。我们可以利用闭包的这一特性来简化函数调用，这里我们看一个具体例子：

复制代码

```
func times(x, y int) int {
  return x * y
}
```

在上面的代码中，times 函数用来进行两个整型数的乘法。我们使用 times 函数的时候需要传入两个实参，比如：

复制代码

```
times(2, 5) // 计算2 x 5
times(3, 5) // 计算3 x 5
times(4, 5) // 计算4 x 5
```

不过，有些场景存在一些高频使用的乘数，这个时候我们就没必要每次都传入这样的高频乘数了。那我们怎样能省去高频乘数的传入呢? 我们看看下面这个新函数 partialTimes：

复制代码

```
func partialTimes(x int) func(int) int {
  return func(y int) int {
    return times(x, y)
  }
}
```

这里，partialTimes 的返回值是一个接受单一参数的函数，这个由 partialTimes 函数生成的匿名函数，使用了 partialTimes 函数的参数 x。按照前面的定义，这个匿名函数就是一个闭包。partialTimes 实质上就是用来生成以 x 为固定乘数的、接受另外一个乘数作为参数的、闭包函数的函数。当程序调用 partialTimes(2) 时，partialTimes 实际上返回了一个调用 times(2,y) 的函数，这个过程的逻辑类似于下面代码：

复制代码

```
timesTwo = func(y int) int {
    return times(2, y)
}
```

这个时候，我们再看看如何使用 partialTimes，分别生成以 2、3、4 为固定高频乘数的乘法函数，以及这些生成的乘法函数的使用方法：

复制代码

```
func main() {
  timesTwo := partialTimes(2)   // 以高频乘数2为固定乘数的乘法函数
  timesThree := partialTimes(3) // 以高频乘数3为固定乘数的乘法函数
  timesFour := partialTimes(4)  // 以高频乘数4为固定乘数的乘法函数
  fmt.Println(timesTwo(5))   // 10，等价于times(2, 5)
  fmt.Println(timesTwo(6))   // 12，等价于times(2, 6)
  fmt.Println(timesThree(5)) // 15，等价于times(3, 5)
  fmt.Println(timesThree(6)) // 18，等价于times(3, 6)
  fmt.Println(timesFour(5))  // 20，等价于times(4, 5)
  fmt.Println(timesFour(6))  // 24，等价于times(4, 6)
}
```

你可以看到，通过 partialTimes，我们生成了三个带有固定乘数的函数。这样，我们在计算乘法时，就可以减少参数的重复输入。你看到这里可能会说，这种简化的程度十分有限啊！

不是的。这里我只是举了一个比较好理解的简单例子，在那些动辄就有 5 个以上参数的复杂函数中，减少参数的重复输入给开发人员带去的收益，可要比这个简单的例子大得多。

小结

好了，今天的课讲到这里就结束了，现在我们一起来回顾一下吧。

在这一讲中，我们讲解了 Go 代码中的基本功能逻辑单元：函数。函数这种语法元素的诞生，源于将大问题分解为若干小任务与代码复用。

Go 语言中定义一个函数的最常用方式就是使用**函数声明**。函数声明虽然形式上与我们之前学过的变量声明不同，但本质其实是一致的，我们可以通过一个等价转换，将函数声明转换为一个以函数名为变量名、以函数字面值为初值的函数变量声明形式。这个转换是你深入理解函数的关键。

我们对函数字面值再进行了拆解。函数字面值是由函数类型与函数体组成的，而函数类型则是由 func 关键字 + 函数签名组成。再拆解，函数签名又包括函数的参数列表与返回值列表。通常我们说函数签名时，会省去参数名与返回值变量名，只保留各自的类型信息。函数签名相同的两个函数类型就是相同的函数类型。

而且，Go 函数采用值传递的方式进行参数传递，对于 string、切片、map 等类型参数来说，这种传递方式传递的仅是“描述符”信息，是一种“浅拷贝”，这点你一定要牢记。Go 函数支持多返回值，Go 语言的错误处理机制就是建立在多返回值的基础上的。

最后，与传统的 C、C++、Java 等静态编程语言中的函数相比，Go 函数的最大特点就是它属于 Go 语言的“一等公民”。Go 函数具备一切作为“一等公民”的行为特征，包括函数可以存储在变量中、支持函数内创建并通过返回值返回、支持作为参数传递给函数，以及拥有自己的类型等。这些“一等公民”的特征，让 Go 函数表现出极大的灵活性。日常编码中，我们也可以利用这些特征进行一些巧妙的代码设计，让代码的实现更简化。

思考题

函数“一等公民”特性的高效运用的例子，显然不限于我们今天提到的这两个，这里我想让你思考一下，你还能列举出其他的高效运用函数“一等公民”特性的例子吗？

欢迎你把这节课分享给更多对 Go 语言的函数感兴趣的朋友。我是 Tony Bai，我们下节课见。

分享给需要的人，Ta订阅后你可得 20 元现金奖励

生成海报并分享

赞 6 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 20 | 控制结构：Go中的switch语句有哪些变化？

下一篇 22 | 函数：怎么结合多返回值进行错误处理？

更多学习推荐

2021 最新大厂
Go 工程师面试真题

大厂面试真题 + 金九银十全新整理 + 核心知识全覆盖

免费领取

精选留言 (11)

写留言

罗杰

2021-11-29

一等公民的概念终于知道从哪里来的了

展开

共 2 条评论 > 3

i6u

2021-11-30

wire 中使用函数作为返回值，统一回收资源也很方便

2

aoe

2021-11-30

partialTimes 的例子就是传说中的柯里化啊

展开

作者回复: 对的。但作为入门课，就不谈那么多概念了。

2

tsingwong
2021-11-30

> 但是像 string、切片、map 这些类型就不是了，它们的内存表示对应的是它们数据内容的“描述符”。当这些类型作为实参类型时，值传递拷贝的也是它们数据内容的“描述符”，不包括数据内容本身，所以这些类型传递的开销是固定的，与数据内容大小无关。
这句话里的 string 也是传递的描述符怎么理解呢？

展开

共 2 条评论 > 1

酥宝话不多
2021-11-30

将函数作为值，就变成了一等值

展开

1

Summer 空城
2021-11-30

老师，您好，请教个问题哈，何时使用静态方法，何时使用结构体的方法呢？我理解需要用到结构体的全局变量的时候担心并发问题，会调用结构体的方法。但是也可以都以参数的形式传递。特别是我们业务代码中的crud，除了接口的具体实现需要结构体，其他的貌似都可以用静态方法，还望老师解惑，多谢老师

展开

作者回复: 你说的“静态方法”是指函数么？

共 5 条评论 > 1

进化菌
2021-11-29

以前一直觉得，函数是公民就是因为用的相当普遍。不过，在go里面，函数充当了太多角色，把面向对象对简化了

展开

1

高雪斌
2021-11-29

觉得在递归等实现上应该有新意，可以精妙的实现一些算法，比如深搜之类的。

1

Geralt of Rivia

2021-11-29

“对于类型为接口类型的形参，Go 编译器会把传递的实参赋值给对应的接口类型形参”，这句话没怎么看明白接口类型形参的特殊性，烦请老师指点一下。

展开

作者回复: 对于类型为接口类型的形参，实参赋值不是简单的值拷贝，而是有一次“装箱”操作，这个操作是go compiler植入的代码完成的。这块还没讲到interface，所以一笔带过了。

共 5 条评论 1

qinsi

2021-12-04

HandlerFunc还有另一半故事，就是函数也可以实现接口

lesserror

2021-12-01

感谢Tony Bai老师这一讲关于函数的内容，全篇都是干货。另外，有几个小问题，麻烦解惑。

1. 命令式语言一般是指哪些语言呢？

...

展开